OysterLoader Unmasked: The Multi-Stage Evasion Loader

By Pierre Le Bourhis

Published: 2026-02-12 · Archived: 2026-04-05 15:27:35 UTC

Introduction

OysterLoader, also known as **Broomstick** and **CleanUp,** is a malware developed in C++, composed of multiple stages, belonging to the loader (*A.k.a.: downloader*) malware family. First reported in June 2024 by Rapid7, it is mainly distributed via web sites impersonating legitimate software which are often IT software for instance: PuTTy, WinSCP, Google Authenticator and Ai software. The loader is primarily employed in campaigns leading to Rhysida ransomware.

According to Expel reports, OysterLoader is used by the Rhysida ransomware group which is closely associated with the WIZARD SPIDER nebula. Besides, the loader is also used to distribute commodity malware such as **Vidar**, the most widespread infostealer by January 2026. According to Huntress, OysterLoader is also distributed via **Gootloader**. Based on our observations and other reports on this threat, it is unclear whether the malware is proprietary to Rhydida ransomware group and friends or sold as **MaaS** on private marketplaces.

Since its apparition, the malware's code has evolved, and analysis by various security vendors highlighted some regressions between its first and current versions, particularly in Command-and-Control (C2) content and in code obfuscation.

Malware Analysis

The malware is distributed mainly through fake websites that copy legitimate software. It is disguised as a software installer and serves as a **MicroSoft Installer** (MSI). The MSI is often signed to appear benign. The infection chain is composed of **four stages**:

1. Stage 1 – Packer [*TextShell*]
2. Stage 2 – Custom shellcode [*TextShell*]
3. Stage 3 – Intermediate DLL acting as a downloader
4. Stage 4 – OysterLoader core

Stage 1 – Obfuscator

The main function of the packer is to load in memory the next stage that is stored "shuffled". According to Huntress' report on Gootlader that mentions OysterLoader, the packer (also called obfuscator in the report) is another malware named **TextShell**.

To load it, it allocates a memory area with necessary permissions (Read, Write, Execute) and makes a raw copy of the data in the newly allocated memory. The copy is made in a bunch of eight bytes. Besides, the code of the initial stage is full of useless API calls to legitimate DLL, the objectives being to avoid execution in particular environments. The packer also embedded a simple anti-debug trap that is present multiple times in the packer.

Besides, the first stage employs common techniques such API hammering and dynamic API resolution.

Legitimate API calls flooding-hammering

In order to hide its malicious code, the loader attempts to make hundreds of calls to legitimate DLLs. In malware, these calls often serve **no operational purpose**. They don't meaningfully change the environment. They look legitimate, though, and that's the point. Their purposes are various:

- Break heuristic detectors "lots of GDI calls – mimikate graphics tool).
- Distract reverse-engineers during static analysis.
- Mislead sandboxes (some sandboxes don't fully emulate GDI calls).
- Introduce chaotic paths in decompiled code.

The malware code is encapsulated with legitimate calls, sometimes, only the prologue of the function is filled with DLLs calls, sometimes the epilogue also contains these patterns.

```
 RevokeDragDrop(0);
DC = GetDC(0);
SolidBrush = CreateSolidBrush(0x75051u);
UnrealizeObject(SolidBrush);
SetMapMode(DC, 2);
v2 = CreateSolidBrush(0xFACB9Fu);
```

```
UnrealizeObject(v2);
SetCommBreak(0);
ptr_buff_struct_mem = allocate_mem;
*(_DWORD *)&allocate_mem->blob0[0x1AD5B] = 0xE539234E;
OaBuildVersion();
OaBuildVersion();
*(_DWORD *)&ptr_buff_struct_mem->blob0[0x1A0D8] = 0x2AD4A690;
SetBkColor(DC, 0x6C072Au);
SetMapMode(DC, 4);
if ( IsDebuggerPresent() )
{
while ( 1 );
}
v4 = CreateSolidBrush(0x202417u);
UnrealizeObject(v4);
*(_QWORD *)&ptr_buff_struct_mem->blob0[0x1A6B9] = 0x475717F412B7ABBCLL;
*(_QWORD *)&ptr_buff_struct_mem->blob0[0x1A6C1] = 0xBA8581BA8183F21BuLL;
*(_QWORD *)&ptr_buff_struct_mem->blob0[0x1A6C9] = 0x3A48C871C86971C8LL;
...
```

Code 1. Extract of decompiled code employing API flooding

In the above example, the only relevant code are (highlighted in blue):

1. Accessing a global variable (here allocate_mem)
2. Copying data at specifics offset (`DWORD` and `QWORD` value)

A more intentional anti-analysis mechanism is the **IsDebuggerPresent() check**; if a debugger is detected, the malware enters an infinite loop (`while(1);`), effectively freezing execution and preventing dynamic analysis. This kind of anti-debugging trick is **far from being advanced**—analysts can easily bypass it by patching the `IsDebuggerPresent` function in *kernel32.dll*, for example by replacing its epilogue with `xor eax, eax; ret` , forcing it to always report that no debugger is attached. Together, these techniques illustrate how malware authors pad their binaries with noise and simple anti-debugging traps to slow down analysts and automated detection systems.

To ease the analysis of this stage, the following script was used to clean up calls to legitimate DLLs that are irrelevant regarding the malware behaviour.

Dynamic API resolution

Dynamic API resolution often implemented through custom hashing algorithms is a widespread technique in modern malware, allowing threats to hide their real API dependencies and evade static detection. The packer is no exception: it relies on dynamically imported APIs, but **each sample uses a slightly different hashing algorithm**, adding variability that complicates signature-based analysis. This lightweight yet effective obfuscation method remains a staple across many malware families.

For instance, the first stage analysed in this report uses the following algorithm, which computes a hash by iterating over each character of an API name and updating a 32-bit accumulator with the formula **h = (h * 0x2001 + ord(ch))**. The following Python script can be used to identify the function corresponding to the hash in the sample.

```
 import pefile

# OysterLoader only needs ntdll and kernel32
dlls_path = [
r"C:\Windows\System32\kernel32.dll",
r"C:\Windows\System32\ntdll.dll"
]

exports = []
for pe in map(lambda dll: pefile.PE(dll), dlls_path):
         for exp in pe.DIRECTORY_ENTRY_EXPORT.symbols:
             if exp.name:
                 exports.append(exp.name.decode('utf-8'))

def hash_function(name: str) -> int:
    h = 0
    for ch in name:
        h = (h * 0x2001 + ord(ch)) & 0xFFFFFFFF
    return h
```

```
hashes = [0x9866A947, 0x895E0804, 0xEA1023BE, 0x8F1E88B1, 0x5CD5A5AA]

for h in hashes:
        for name in exports:
            if hash_function(name) == h:
                print(f"[+] Match found: {name} for hash: 0x{h:x}")
                break
```

Code 2. Python snippet to resolve the API function

```
In [6]: hashes = [0x895E0804, 0xEA1023BE, 0x8F1E88B1]

In [7]: for h in hashes:
   ...:     for name in exports:
   ...:         if hash_function(name) == h:
   ...:             print(f"[+] Match found: {name} for 0x{h:x}")
   ...:
[+] Match found: RtlInitUnicodeString for 0x895e0804
[+] Match found: LdrLoadDll for 0xea1023be
[+] Match found: LdrGetProcedureAddress for 0x8f1e88b1
```

Figure 1. Output of the Python snippet used to resolve API hash

Initial stage workflow

The main component of the first stage of the loader works as follows:

1. Dynamically resolves the following function using a custom hashing algorithm `NtAllocateVirtualMemory`, `LdrGetDllHandle`, `RtlInitUnicodeString`, `LdrLoadDll`, `LdrGetProcedureAddress`.
2. Uses `NtAllocateVirtualMemory` to allocate memory with `RWX` permissions.
3. Copies data into the allocated buffer.
4. Dynamically resolves additional functions using uniquely the pair of functions from ntdll: `LdrLoadDll`, `LdrGetProcedureAddres`. This pair is used to load the following functions: `LoadLibrary`, `GetProcAddress`, `ExitProcess`, `VirtualProtect`, `exit`, `ShowWindow`, `InternetOpenW` that will be necessary for next stages.
5. Executes a specific fixed offset in the shellcode previously allocated.

Note that the structure of this initial stage is pretty straightforward regardless of the numerous noisy DLL calls flooding the binary.

In this primary stage, the loader saves particular information in a dedicated structure that we named core for this analysis. The core structure that will also be used by the next stage has this composition:

```
struct core
{
uint8_t compressed_data[136512];
_QWORD entypoint;
_BYTE config[6940];
_QWORD LoadLibraryA;
_QWORD ExitProcess;
_QWORD GetProcAddress;
_QWORD VirtualProtect;
_QWORD unknown0;
_QWORD exit;
_BYTE flags[8];
_QWORD ShowWindow;
_QWORD unknown1;
_QWORD InternetOpenW;
};
```

Code 3. Internal C structure of the stage 1 of OysterLoader

Stage 2 – Shellcode

The code executed at this stage is as shellcode, though extremely dependent on the previous stage as it requires access to the core structure mainly to access the `LoadLibrary` and `GetProcAddress` couple to dynamically load another bunch of functions. It needs to have the offset of a blob of data (named compressed_data in the core structure).

Once executed, the stage-2 shellcode immediately transfers control to a bespoke **LZMA decompression routine**, comprising a substantial part of the payload's codebase. The implementation clearly mirrors the standard LZMA range decoder: it maintains a large set of bit-probability models, performs the characteristic 11-bit probability updates, and

transitions through the well-known LZMA state machine to distinguish literal bytes from matches and repeated match lengths.

After reconstructing the original payload, the shellcode performs a pass of **relocation fixups**, scanning the decompressed buffer for relative `CALL` (E8) and `JMP` (E9) opcodes and rewriting their offsets to absolute addresses-an essential adaptation for position-independent code running at unpredictable memory locations. With the payload's code layout recovered, the shellcode proceeds to **resolve its imports dynamically** by manually calling `LoadLibraryA` and `GetProcAddress`, populating an ad-hoc import table through function pointers stored in the staged data region. These resolved APIs are then used to adjust **memory protections** via `VirtualProtect`, clearing non-executable bits and transitioning the decompressed region to an executable state. Finally, with all dependencies resolved and memory prepared, the shellcode transfers control to the newly reconstructed payload, invoking its entry point and continuing execution into the next stage of the malware.

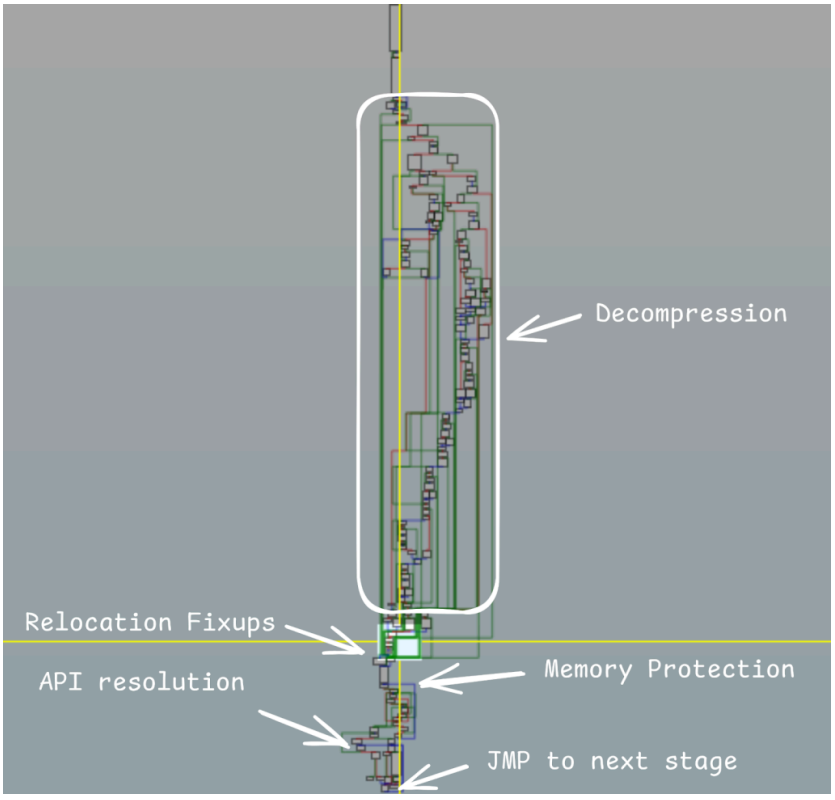

Figure 2. OysterLoader stage 2 shellcode graph overview

A custom Python script has been used to inflate the LZMA compressed buffer; a script is available on this Gist.

OysterLoader's use of a **custom LZMA implementation** rather than standard compression tools serves multiple strategic purposes that are characteristic of good malware development skills. While the compression parameters (lc=3, lp=0, pb=2) remain standard for optimal performance, the custom header format and modified bitstream prevent automated analysis by common tools like 7-Zip, xz-utils, or Python's lzma module.

The custom header structure (storing properties at specific blob offsets rather than using the standard 13-byte LZMA header) also serves an evasion purpose – signature-based detection systems looking for standard LZMA magic bytes (0x5D for properties, or .xz/.lzma file signatures) will fail to identify this as compressed data. Additionally, the non-standard bitstream format means that even with a reconstructed proper LZMA header with correct parameters, standard decompression will fail due to subtle modifications in the probability models or range decoder implementation.

Once decompressed, the shellcode also copies some addresses (for instance DLL functions that were loaded in the previous stage) and that will be used by stage 3.

Stage 3 – Downloader

This third stage is composed of a set of common functions going from language verification, keyboard layout identification to anti-debug, assembly un-alignment to trick decompiler. However, this is the pre-final stage, it acts as an environment test, this is also the first time the malware communicates with a server.

Stage 3 – Environment verification

An interesting function remains in the sample, even though it is never invoked. It was likely used during development and may have been inadvertently left in the final build. The function checks whether the host executing the malware has its system language set to **Russian**.

Hook verification, by executing fourteen times a Beep for 2 seconds followed by Sleep during 4.5 seconds, at the end of this loop, the function computes the saved time (before beep, sleep loop) and the current time to calculate the elapsed time of sleep. Overall, the function acts as a **timing measurement wrapper** around repeated sleep intervals, likely used for debugging, delay insertion, or basic anti-analysis timing checks.

The main function uses `EnumProcess` to count the number of **running processes**, if the count is below **60**, the malware exits. Otherwise it creates a mutex `h6p#dx!&fse?%AS!` , the prefix value appears to be shared among different versions and builds of the loader.

- VirusTotal: `behaviour:"h6p#dx!&fse?%AS!"` (First tracked campaign)
- VirusTotal: `behaviour:"s6p1dx!&fse?%AS!"` (New campaign November 2025)
- VirusTotal: `behaviour:"dx!&fse?%AS!"` (Generic approach)

Stage 3 – C2 communication

Once these preliminary actions are validated, the malware starts interacting with the first layer of C2 servers. It communicates over HTTPS, the first message acts as a registration as indicated by the C2 URL (`/reg`), additionally the bot generates two random strings composed only of alpha lower, upper cases and digits. The first random is sent in the HTTP header `x-amz-cf-id` this value is used to identify the bot and the second one is sent in the Content-Encoding, this value is used as the botnet or campaign identifier. For this first request the User-Agent header is set to `WordPressAgent` . The malware disguises its C2 traffic as legitimate HTTP activity using generic paths, spoofed headers such as `Content-Encoding` and `x-amz-cf-id` , and a fake **WordPressAgent** user-agent to blend in with normal web traffic and evade simple detection by security tools that rely on protocol and behavioral anomalies.

If the text success is received by the infected host, the malware continues its execution. Then the bot sends a second **GET** request on the `/login` endpoint. For this request, the user-agent changed to `FingerPrint` . This is a surprising modification as legitimate software devices do not change this value once the HTTP communication has started.

The C2 responds with an image (*c.f.*: the Content-Type set to `image/x-icon`), the malware uses steganography to hide the next stage payload as an icon. The data are in the following format: the first size byte defines the size of the obfuscated data. Followed by junk data that composed a real image (*c.f. Annex 1 – ico image*). The function searches for the **endico** pattern, once it finds it, itdecrypts the data using **RC4** algorithm along with a hardcoded key in the binary.

The bot expected to find a valid PE from the downloaded data, as it expects to have for the first two bytes the MZ value. Based on the different sample we analysed, the RC4 key remains the same:

```
vpjNm4FDCr82AtUfhe39EG5JLwuZszKPyTcXWVMHYnRgBkSQqxzBfb6m75HZV3UyRY8vPxDna4WC2KMAgJjQqukrFdELXeGNSws9SBFXnYJ6ExMyu97KCebD5mTwaUj42NPAvHdk
```

The following link to Gist hosts a script to locate and decrypt the next stage from the HTTP GET /login response.

Once decrypted the PE is written into a file name `COPYING3.dll` in the `%APPDATA%` directory. The DLL is executed by the task scheduler. The task name **COPYING** runs every 13 minutes and is configured as below:

```
C:\Windows\System32\schtasks.exe /Create /SC MINUTE /MO 13 /TN "COPYING3" /TR
"C:\Windows\System32\rundll32.exe C:\Users\<redacted>\AppData\Roaming\<15 random alphanum>\COPYING3.dll
DllRegisterServer"
```

NB: The name of the dropped DLL and the scheduled task name changed for each version/campaign. In latest campaigns we denote the following names:

- `COPYING3`
- `VisualUpdater`
- `AlphaSecurity`
- `DetectorSpywareSecurity`

Stage 4 – COPYING3 – Core

`COPYING3.dll` is valid PE32+ file, that exposed only two functions: `DllRegisterServer` and the default `DllEntrypoint` , the default entry point looks familiar as the threat actor reused the same layer of obfuscation as described previously for stage 0 of the infection (*e.g.* copy of a shellcode in a specific buffer that executes itself and does the custom LZMA inflation followed by the reallocation fixup before giving the execution flow to the final sequence of code).

An unusual way to execute the last stage is present in OysterLoader infection. In more common situations where DLL is used, the previous stage executes a specific export of the DLL or uses one of the DLL hijacking techniques. However, in this current scenario, the previous stage was previously called the `DllEntrypoint` before the `DllRegisterServer` .

This stage is the core of the malware as its primary behaviour is to interact with the C2 to retrieve additional payload.

C2 interaction

The malware implements a straightforward HTTP-based command and control protocol operating over port 80 without encryption. It maintains connectivity with three hardcoded C2 servers (`85.239.53[.]66` , `51.222.96[.]108` , and `135.125.241[.]45`) using a robust fallback mechanism that cycles through each server with three retry attempts spaced **9 seconds** apart before moving to the next.

The communication follows a dual-endpoint pattern:

1. `/api/kcehc` ("check" spelled backwards) to indicate that a new infected host has been compromised and exfiltrate victim host fingerprint.
2. **POST** requests to `/api/jgfnsfnuefcnegfnehjbfncejfh` that acts as a beacon endpoint where it received C2 orders, status, additional payload, etc.

The loader uses a custom and changing message structure. The first data that is sent to the C2 is a JSON completely encoded with a custom algorithm based on Base64 encoding. The algorithm uses a **random** and **unique shift** value that varies for each message, besides it also uses a **non-default Base64** alphabet order (yog/ `N3fj5ISmbep=Wu2k+BZcP0t4CYR1dQxHUaXEwGDKJV7i9ML6snhzrlqO8vAFT`). The random shift value is generated for each message using Mersenne Twister algorithm.

The non-standard alphabet is hardcoded in the malware and seems to be preserved among OysterLoader versions. To improve the grasp of OysterLoader server communication we develop a Python script that brute force the shifting value to decode the message.

P.S. The reason why the bruteforce attack is efficient here is because of the length of the Base64 alphabet where the developer must ensure the encoding remains functional which consequently requires a shifting value between 1 and 65.

The decoding script is available on this Gist; note that if the Base64 custom alphabet changes in future versions, the script should be updated too. We plug a stdin reader to facilitate a PCAP reader with the script. For the showcases sample of this paper, the following command is used:

tshark -r capture.pcap -T fields -e http.file_data| python com-decoder.py

All traffic uses `WinINet` APIs (`InternetOpenW` , `InternetConnectW` , `HttpOpenRequestW` , `HttpSendRequestA`) with a distinctive User-Agent string `Mozilla/6.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/91.0.4472.124 Safari/537.36` and exchanges data in **JSON** format. The raw JSON only contains a unique key named content which contains the message encoded with the custom Base64 algorithm. The content of the decoded JSON are:

- `/api/kcehc` expected **Request Body (System Info):**

```
 {
"a1": "[timestamp counter]",
"a2": "[Hardcoded value in the binary]",
"a3": "[username]",
"a4": "[computer name]",
"a5": "[domain info]",
"a6": "[os version string]",
"a7": "[domain name]",
"a8": "[hardcoded version number]",
"a9": "[PE type: EXE or DLL]",
"a10": "[Windows Version]"
}
```

- `/api/jgfnsfnuefcnegfnehjbfncejfh` **Request Body (Status Beacon):**

```
 {
"b1": timestamp counter,
"b2": random token,
"b3": status message,
"b4": execution result
}
```

NB: In the above JSON schema, all fields are not required for each message.

C2 response when a new payload is to execute:

```
 {
"r1": command type,
"r2": payload file name,
"r3": execution parameter flag,
"r4": additional config flag,
"r5": payload data encoded in standard Base64
}
```

C2 endpoint and protocol update

In the latest bot version, the JSON fingerprint format has been overhauled, to include running-process data. The original keys (`a1` – `a10`) are now `t1` – `t12` : `t1` – `t10` retain their previous meanings, while `t11` and `t12` carry arrays of process names and their Process Identifier (PID) (*c.f.: Annex 2 – Example of exfiltrated fingerprint*).

At the same time, the Base64 alphabet used for encoding shifts dynamically during C2 exchanges. Although the bot still starts with its original shuffle key to encode the outbound fingerprint, the C2's JSON response adds a new field, `tk` , which provides an updated replacement alphabet. From that point forward, the bot applies this new `tk` string in its decoding routine for all subsequent communications.

Once decoded the returned JSON from the **POST** request to /api/v2/facade is as follows:

```
 {
"ti": identifier,
"tk": the new Base64 substituion alphabet,
"tr": the final URL resource to beacon, eg: `/api/v2/<tr>,
"tt": ""
}
```

In the new version, a third endpoint has been added. Previously, the bot used two URLs—one for pings and fingerprint exfiltration and another for beaconing. Now the initial phase is split across two endpoints: it first issues an empty **GET** request to `/api/v2/init` , then sends the encoded fingerprint via **POST** to `/api/v2/facade` . The original beaconing URL is now defined in the response of the `/api/v2/facade` by the JSON key tr.

The new URLs path are:

- `/api/v2/init` (*check*)
- `/api/v2/facade` (*send fingerprint and receive new Base64 alphabet*)
- `/api/v2/YgePIY5zPSoGUjzRx7C50MTx6EzABXIPd` (*beacon*)

Hunting – Infrastructure

OysterLoader malware operates with a two-tiered server infrastructure.

1. The initial layer, the delivery server, handles the initial connection via the /reg and /login URLs. Its primary role is to host the steganographically concealed next stage of the malware. This stage has shown consistency, with the developer maintaining the same communication patterns and URLs.
2. The second layer functions as the final C2 server. This server is responsible for victim interaction, including collecting information and issuing commands and orders. Unlike the delivery stage, the C2 server has been updated twice since the initial report on OysterLoader.

Date	URL
First observed version May 2024 to October 2025	`/api/connect` `/api/session`
June 2025 to September 2025	`/api/kcehc` `/api/jgfnsfnuefcnegfnehjbfncejfh`

December 2025	`/api/v2/init`
	`/api/v2/facade`
	`/api/v2/YgePIY5zPSoGUjzRx7C50MTx6EzABXIPd`
	(also `/api/v2/1X54xw9ocWxlx9p2VWGxZ4ljAtr`)

As of January 2026, the latest OysterLoader C2 are:

```
 hxxps://grandideapay[.]com/api/v2/facade
hxxp://nucleusgate[.]com/api/v2/facade
hxxps://cardlowestgroup[.]com/api/v2/facade
```

```
hxxps://socialcloudguru[.]com/api/v2/facade
hxxps://coretether[.]com/api/v2/facade
hxxps://registrywave[.]com/api/v2/facade
```

Conclusion

OysterLoader, also known as Broomstick and CleanUp, is an sophisticated, multi-stage malware loader developed in C++ that continues to pose a relevant threat into early 2026. Its primary objective is to facilitate malicious campaigns, notably those conducted by the **Rhysida ransomware** group, and to distribute commodity malware like **Vidar** in a business model that remains unclear at the time of writing this report.

The detailed analysis presented in this report highlights a comprehensive set of techniques employed by the loader across its four stages, designed specifically to ensure persistence and evade detection:

- **Evasion & Obfuscation:** The initial stage leverages excessive legitimate API call hammering and simple anti-debugging traps to thwart static analysis. Furthermore, dynamic API resolution employs a custom hashing algorithm that varies between samples, complicating signature-based detection.
- **Stealthy Delivery:** The core payload is delivered in a highly obfuscated manner. Stage 2 utilises a custom LZMA decompression routine with a non-standard header and modified bitstream, effectively bypassing common decompression and analysis tools.
- **Advanced C2 Communication:** The final stage implements a robust C2 communication protocol that features a dual-layer server infrastructure and highly-customized data encoding. It uses a non-default Base64 alphabet with a unique, random shift value for each message, making traffic analysis and automated decoding challenging. The malware also employs techniques to blend in, such as spoofing HTTP headers and user-agents, while environment checks (like process counting) act as an additional layer of anti-analysis protection.

The constant evolution in OysterLoader's code, including updated C2 endpoints and JSON fingerprinting schemas, signals the high level of activity and commitment from the threat actors. The quality and complexity of the malware's development strongly suggest that OysterLoader will remain a significant and persistent threat in the near term.

To protect our customers from OysterLoader, Sekoia.io analysts will continue to proactively monitor this threat.

Annexes

Annex 1 – ico image

```
(.venv) → scripts exiftool -FileSize -FileType -FileTypeExtension -MIMET[2/41768]
eCount -ImageWidth -ImageHeight -ImageLength -ImageSize ../login-ico.data
File Size                       : 724 kB
File Type                       : ICO
File Type Extension             : ico
MIME Type                       : image/x-icon
Image Count                     : 9
Image Width                     : 16
Image Height                    : 16
Image Length                    : 1128
Image Size                      : 16x16
(.venv) → scripts python extract-ico.py ../login-ico.data

[+] Encrypted size: 291496 bytes
[+] Decrypted payload written to payload.bin
[+] Valid PE file detected!
0000  4d 5a 90 00 03 00 00 00 04 00 00 00 ff ff 00 00   MZ..............
0010  b8 00 00 00 00 00 00 00 40 00 00 00 00 00 00 00   ........@.......
0020  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00   ................
0030  00 00 00 00 00 00 00 00 00 00 00 00 f0 00 00 00   ................
0040  00 00 00 00 00 00 00 00 00 00 00 00 00 00 21 54 68   ..............!Th
0050  69 73 20 70 72 6f 67 72 61 6d 20 63 61 6e 6e 6f   is program canno
0060  74 20 62 65                                       t be

... <redacted> ...
```

```
→ Broomstick viu login-ico.data
```

Annex 2 – Example of exfiltrated fingerprint

```
{   't1': '',
   't10': 'Windows 10 Pro | 19044.1288',
   't11': '7-Zip 23.01 (x64) | Mozilla Firefox (x64 en-US) | Mozilla '
          'Maintenance Service | Node.js | Microsoft .NET Host - 8.0.3 (x64) '
          '| Microsoft ASP.NET Core 8.0.3 Targeting Pack (x64) | Microsoft '
          '.NET Targeting Pack - 8.0.3 (x64) | '
          'Microsoft.NET.Sdk.Maui.Manifest-8.0.100 (x64) | Microsoft .NET '
          'Toolset 8.0.203 (x64) | Microsoft Visual C++ 2022 X64 Additional '
          <REDACTED>

          '.NET Host FX Resolver - 9.0.1 (x64) | Microsoft .NET Host FX '
          'Resolver - 8.0.3 (x64) | Microsoft.NET.Sdk.Aspire.Manifest-8.0.100 '
          '(x64) | ',
   't12': '[System Process] (PID: 0) | System (PID: 4) | Registry (PID: 100) '
          '| smss.exe (PID: 344) | csrss.exe (PID: 436) | wininit.exe (PID: '

                <REDACTED>

          '(PID: 4744) | SppExtComObj.Exe (PID: 3160) | svchost.exe (PID: '
          '3696) | svchost.exe (PID: 1560) | svchost.exe (PID: 5548) | '
          'sysmon.exe (PID: 6956) | unsecapp.exe (PID: 6992) | svchost.exe '
```

```
            '(PID: 6544) | vt-windows-event-stream.exe (PID: 5672) | '
            'vt-windows-event-stream.exe (PID: 5660) | '
            'vt-windows-event-stream.exe (PID: 5324) | conhost.exe (PID: 5320) '
            '|
    't2': '',
    't3': 'Bruno',
    't4': 'DESKTOP-ET51AJO',
    't5': '',
    't6': '',
    't7': 'WORKGROUP',
    't8': '1',
    't9': 'dll'
}
```

The complete JSON is available on this gist.

CTI Cybercrime Malware

Share this post:
